Runtime randomization and perturbation for virtual machines.

JAVIER CABRERA ARTEAGA

Licentiate Thesis in [Research Subject - as it is in your ISP]
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT XXXX:XX
ISBN XXX-XX-XXXX-XXX-X

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i [ämne/subject] [veckodag/weekday] den [dag/day] [månad/month] [år/2022] klockan [tid/time] i [sal/hall], Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

Write your abstract here...
Keywords: Keyword1, keyword2, ...

Sammanfattning

Write your Swedish summary (popular description) here...
Keywords: Keyword1, keyword2, ...

Acknowledgements

Write your professional acknowledgements here...

Acknowledgements are used to thank all persons who have helped in carrying out the research and to the research organizations/institutions and/or companies for funding the research.

Name Surname,
Place, Date

<a href="https://www.flaticon.es/iconos-gratis/personalizado" title="personalizado iconos">Personalizado iconos creados por monkik - Flaticon</a>

<a href="https://www.flaticon.es/iconos-gratis/computadora" title="computadora iconos">Computadora iconos creados por Freepik - Flaticon</a>

Contents

Chapter 1

Introduction

TODO Two references per paragraph

TODO Base paragraph by the papers

Internet of Software Monoculture

Software Diversification as a solution

TODO Moved from Chapter 2 The low presence of defenses implementations for WebAssembly motivates our work on Software Diversification as a preemptive technique that can help against known and yet unknown vulnerabilities.

Artificial Software Diversification

Fine-grained Diversification

1.1 Thesis Statement

1.2 Research questions

1. RQ1. To what extent can we artifically generate program variants for WebAssembly ? TODO Motivation

2. RQ2. To what extent are the generated variants dynamically different? TODO Motivation

3. RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms? TODO Motivation

1.3 Contributions

The contributions of this thesis are:

(C_1) Methodological contribution: TODO

(C_2) Experimental contribution: TODO

(C_3) Theoretical contribution: TODO

(C_4) Technical contribution: TODO TODO Point to chapter 3

1.4 Publications

(P_1) Bytecode analysis for V8 and DTW: TODO

(P_2) Superoptimization: TODO

(P_3) CROW: TODO

(P_4) MEWE: TODO

Origin of contributions ?

Other publications and talks

1. wasm-mutate: TODO

Thesis layout

Chapter 2

Background & State of the art

This chapter discusses state of the art in the areas of *WebAssembly* and *Software Diversification*. In Section 2.1 we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, language specification, and security-related issues. In Section 2.2, we present a summary of Software Diversification, its foundational concepts and highlighted related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. In Section 2.3, we finalize the chapter by stating our novel contributions and comparing them against state-of-the-art related works.

2.1 WebAssembly overview

Over the past decades, JavaScript has been used in the majority of the browser clients to allow client-side scripting. However, due to the complexity of this language and to gain in performance, several approaches appeared, supporting different languages in the browser. For example, Java applets were introduced on web pages late in the 90's, Microsoft made an attempt with ActiveX in 1996 and Adobe added ActionScript later on 1998. All these attempts failed to persist, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Emscripten proposed with a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript itself. Asm.js was first implemented as an LLVM backend. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [39] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline or those that can be directly translated from the source code. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization. The work of Van Es et al. [32] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Despite encouraging results, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and the recompilation of the JavaScript code which implies significant overhead.

Following the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially stated later by the work of Haas et al. [31] in 2017. The announcement of WebAssembly marked the first step of standarizing bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [31]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

From source to Wasm

All WebAssembly programs are compiled ahead-of-time from source languages. LLVM includes Wasm as a backend since version 8.0.0, supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[4]. The resulting binary, works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 2.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations. For example, if the Wasm binary is generated out of the LLVM pipeline, all optimizations in the LLVM

The step ② builds the standard library for Wasm usually as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

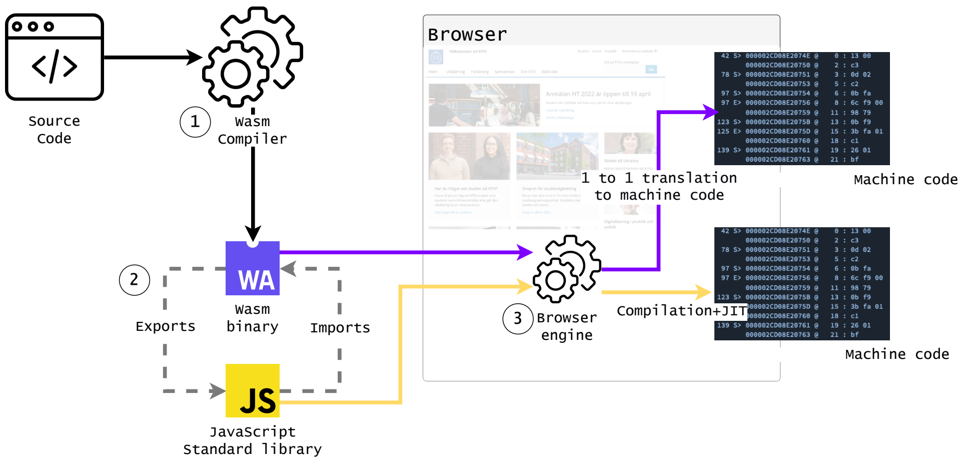
[1]`https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2]`https://satoshinm.github.io/NetCraft/`

[3]`https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

[4]subset of the TypeScript language

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes the compilation process, the generated machine code for Wasm is final and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions.



```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
125 E> 000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

```
 42 S> 000002CD08E2074E @    0 : 13 00
       000002CD08E20750 @    2 : c3
 78 S> 000002CD08E20751 @    3 : 0d 02
       000002CD08E20753 @    5 : c2
 97 S> 000002CD08E20754 @    6 : 0b fa
 97 E> 000002CD08E20756 @    8 : 6c f9 00
       000002CD08E20759 @   11 : 98 79
123 S> 000002CD08E2075B @   13 : 0b f9
125 E> 000002CD08E2075D @   15 : 3b fa 01
       000002CD08E20760 @   18 : c1
139 S> 000002CD08E20761 @   19 : 26 01
       000002CD08E20763 @   21 : bf
```

Figure 2.1: WebAssembly building, compilation in the host engine and execution.

Wasm can execute directly and is platform independent, making it useful for IoT and Edge computing [2, 16]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly . In this case, the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the Bytecode Alliance [5] proposed WebAssembly System Interface (WASI) [9]. WASI is the foundation to build Wasm code outside of the browser with a POSIX system interface platform. It standarizes the adoption of WebAssembly outside web browsers [18] in heterogeneous platforms.

WebAssembly specificities

WebAssembly defines its own Instruction Set Architecture (ISA) [28]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary

[5] https://bytecodealliance.org/

components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types, each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 3.3 and Listing 2.2 we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 2.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32`, `i64`, `f32` and `f64`. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way.A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++; however, the compiler should populate the static table of functions.

In Wasm, all instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 2.2. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes

Listing 2.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.2: WebAssembly code for Listing 3.3.

```
1  (module
2    (type (;0;) (func (param f32) (result i32)))
3    (type (;1;) (func))
4    (type (;2;) (func (result i32)))
5    (import "env" "ftoi" (func $ftoi (type 0)))
6    (func $main (type 2) (result i32)
7      (local i32 i32)
8      i32.const -1000
9      local.set 0
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

independently and cannot execute or refer to outer block values. This is quarantieed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in Listing 2.2, Line 19 forces the execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

We want to remark that de description of Wasm in this section follows the version 1.0 of the language and not its proposals for extended features. We follow those features implemented in the majority of the vendors according to the Wasm roadmap [29]. On the other hand we excluded instructions for datatype conversion, table accesses and the majority of the arithmetic instructions for the sake of simplicity.

WebAssembly's security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [17] and makes easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, WebAssembly is vulnerable under certain conditions, at the execution engine's level [22]. Implementations in both browsers and standalone runtimes [2] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [26]. One of our previous contributions trigger a CVE[6] on the code generation component of wasmtime, highlighting that even when the language specification is meant to be secure, the underlying host implementation might not be. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [14] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly . Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors.

2.2 Software Diversification

Software Diversification has been widely studied in the past decades. This section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey of Baudry and Monperrus [38], the motivation for software diversification can be separated in five categories: reusability [58], software testing [49], performance [46], fault tolerance [68] and security [65]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they use the generated diversification.

There are two primary sources of software diversification: Natural and Artificial Diversity [38]. This work contributes to the state of the art of Artificial Diversity, which consists of artificially synthesizing software. We have found that the foundation for artificial software diversity has barely changed since Cohen in 1993 [65]. Therefore, the work of Cohen is the cornerstone of this dissertation.

[6] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

Variants' generation

Cohen et al. proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen et al. proposed 10 concrete transformation strategies that can be applied at fine-grained level. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry and Monperrus [38] and the work of Jackson et al. [44], in 5 strategies.

(S1) *Equivalent instructions replacement* Semantically equivalent code can replace pieces of programs. For example, it replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result(garbage instructions). There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [45] and Homescu et al. [42] introduce the usage of inserting NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples or constructs all possible programs for a specific language. In this topic, Jacob et al. [52] proposed the technique called superdiversification for x86 binaries. Similarly, Tsoupidi et al. [11] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

(S2) *Instruction reordering* This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers resort them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [62, 59] proposed the random permutation of the order of variables and routines for ELF binaries.

(S3) *Adding, changing, removing jumps and calls* This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [65] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [66] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [37] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [59] extended their previous approach [62], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [1] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

(S4) *Program memory and stack randomization* This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [62, 59] also proposed to randomize the base addresses of applications and the library memory regions, and the random introduction of gaps between memory objects in ELF binaries. Tadesse Aga and Autin [21] and Lee et al. [3] recently proposed a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [56] proposed to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [10] transform programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

(S5) *ISA randomization and simulation* This strategy encodes the original program binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving the examination of code. Kc et al. [61] and Barrantes et al. [63] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [64] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [33] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL.

Variants' equivalence

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [41] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent. This process is undecidable; still, some approaches approximate it [20]. Cohen [65] simplifies this process by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The easiest way to guarantee the equivalence property is by construction. For example, in the case illustrated in S1 for Cleemput et al. [45] and Homescu et al. [42], the transformation strategy is designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [15], it is equivalent to the original. However, it is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT solvers) [53] or fuzzers [?] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical expressions. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two math expressions return a different output. The main limitation of this technique is that all algorithms cannot be translated to a math expression, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [?] . On the other hand, fuzzers look randomly for inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not semantically equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand. In practice, a fuzzer needs a significant amount of time to check two programs, and this time takes hours at the minimum.

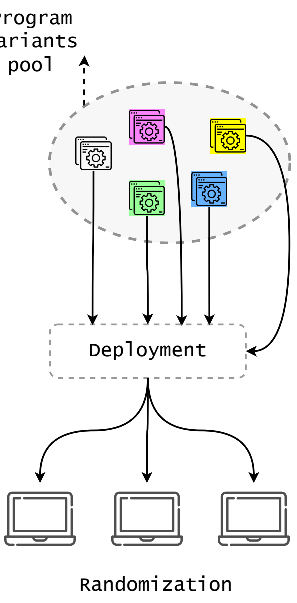
We have found that SMT solvers are the best option for checking most of the fine-grained transformations previously mentioned for two main reasons. First, the field of SMT is mature and provides battle-tested tooling [53, 25]. Second, the existing SMT tooling is configurable and flexible, making the checking of program variants feasible in terms of the SMT solver execution time. In Chapter 3 we describe how we apply SMT solvers in our contributions for equivalence checking.

Usages of Software Diversity

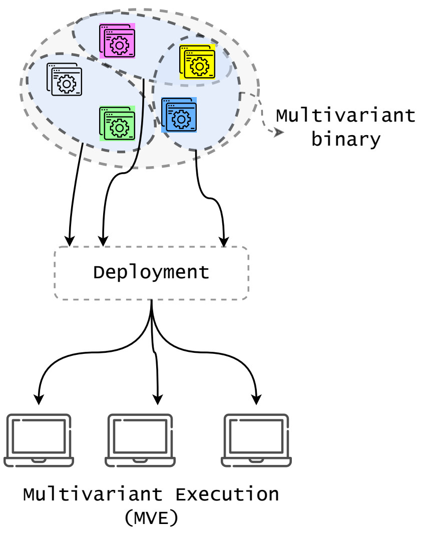
After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution(MVE) [44]. In Figure 2.2a and Figure 2.2b we illustrate both scenarios.

(U1) *Randomization:* In the first scenario Figure 2.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [44] call this a herd immunity scenario since vulnerable binaries can affect only part of the client's community. Chew and Song [64], Kc et al. [61] and Xu et al. [10] recompile the Linux Kernel to achieve this scenario. The Polyverse company [7] materialize their ideas in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level. Similarly, El-Khalil and colleagues [60] proposed to use a custom compiler to generate different binaries out of the compilation process. On the same topic, Aga and colleagues [21] proposed

[7]`https://polyverse.com/`



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.



(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed per. Each deployment randomly selects which multivariant binary is assigned to each host. A variant from the multivariant binary is randomly executed at runtime in the host, .

Figure 2.2: Software Diversification usages.

to generate program variants by randomizing its data layout in memory, making each variant to access the same memory in different ways.

(U2) *Multivariant Execution(MVE):* In the second scenario Figure 2.2b, multiple program variants are composed in one single binary (multivariant binary) [57]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime. The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [62]. Bruschi et al. [55] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [54] modified a standard library that generates 32-bit Intel variants where the stack grows in the opposite direction, checking for memory inconsistencies. Notably, Davi et al.

proposed Isomeron [36], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. Neatly exploiting the limit case of executing only two variants in a multivariant binary has demonstrated to harden systems [50, 43, 35, 23, 36]. Further two variants per multivariant binary, Agosta et al. [40] and Crane et al. [37] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling know vulnerabilities such as (JIT)ROP attacks [47] and power side-channels [48]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [44]. Our work contributes to both scenarios as a preemptive technique for hardening WebAssembly .

2.3 Statement of Novelty

We contribute to Software Diversification for WebAssembly using Artificial Diversification, for Randomization and Multivariant Execution usages (U1, U2). In Table 2.1 we listed related work on Artificial Software Diversification that support our work. The table is composed by the authors and the reference to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarize their technical contribution and the reach of their work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

Our first contribution, CROW [8] generates multiple program variants for WebAssembly using the LLVM pipeline. It contributes to state of the art in artificially creating randomization for WebAssembly (U1). Because of the specificities of code execution in the browser (mentioned in Section 2.1), this can be considered a randomization approach. For example, since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW. With MEWE [7], our second contribution, we randomly select from several variants at runtime, creating a multivariant execution scheme(U2) that randomizes the observable behaviors at each run of the multivariant binary.

Conclusions

In this chapter, we presented the background on the WebAssembly language, including its security issues and related work. This chapter aims to settle down the foundation to study automatic diversification for WebAssembly . We highlighted related work on Artificial Software Diversification, showing that it has been widely researched, not being the case for WebAssembly . On the other hand, current available implementations for Software Diversification cannot be directly ported to Wasm. We placed our contributions in the field of artificial diversity. In Chapter 3 we describe the technical details that lead our contributions. Besides, the impact of our contributions is evaluated by following the methodology described in Chapter 4.

Authors	**S1**	**S2**	**S3**	**S4**	**S5**	**U1**	**U2**	**Main technical contribution**
Pettis and Hansen [66]		✓		✓		✓		Custom Pascal compiler for PA-RISC architecture
Chew and Song [64]			✓			✓		Linux Kernel recompilation.
Kc et al. [61]					✓			Linux Kernel recompilation.
Barrantes et al. [63]					✓	✓		x86 to x86 transformations using Valgrind
Bhatkar et al. [62]	✓	✓		✓		✓		ELF binary transformations
El-Khalil and Keromytis [60]						✓		custom GCC compiler for x86 architecture
Bhatkar et al. [59]	✓	✓		✓		✓		C/C++ source to source transformations and ELF binary transformations
Younan et al. [56]				✓				custom GCC compiler
Bruschi et al. [55]				✓		✓		ELF binary transformations.
Salamat et al. [54]			✓				✓	Custom GNU compiler
Jacob et al. [52]	✓	✓						x86 to x86 transformations
Salamat et al. [50]				✓			✓	x86 to x86 transformations
Amarilli et al. [48]	✓				✓	✓		Polymorphic code generator for ARM architecture
Jackson [44]	✓					✓	✓	LLVM compiler, only backend for x86 architecture
Cleemput et al. [60]	✓					✓		x86 to x86 transformations
Homescu et al. [42]	✓					✓		LLVM 3.1.0†
Crane et al. [37]	✓	✓	✓				✓	LLVM, only backend for x86 architecture
Davi et al. [36]							✓	Windows DLL instrumentation
Couroussé et al. [33]	✓	✓			✓	✓		Custom GCC compiler targeting micro-controllers
Lu et al. [23]				✓			✓	GNU assembler for Linux kernel
Belleville et al. [27]	✓			✓		✓		Only C language frontend, LLVM 3.8.0†
Aga et al. [21]				✓		✓		Data layout randomization, LLVM 3.9†
Österlund et al. [19]				✓			✓	Linux Kernel recompilation.
Xu et al. [10]				✓		✓		Custom kernel module in Linux OS
Lee et al. [3]				✓		✓		LLVM 12.0.0 backend for x86
Cabrera Arteaga et al. [8]	✓	✓	✓	✓		✓		Any frontend language for LLVM version 12.0.0 targeting Wasm backend
Cabrera Arteaga et al. [7]	✓	✓	✓	✓			✓	Any frontend and backend language for LLVM version 12.0.0

† Notice that LLVM only supports WebAssembly backend from version 8.0.0

Table 2.1: The table is composed by the authors and the reference to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarize their technical contribution. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. The last two rows locate our contributions.

Chapter 3

Technical contributions

We aim to create Artificial Software Diversity for WebAssembly , by providing tools to make the process easier and feasible for developers and researchers. As far as we know, there is no software that provides Artificial Software Diversification for WebAssembly. Therefore, we need to enunciate the engineering foundation to implement the strategies in Section 2.2. Our implementations are part of the contributions of this thesis. Concretely, we provide two software artifacts that complement this work. Our approach generate WebAssembly program variants statically at compile time to provide randomization. Besides, it provides the tooling to generate MVE binaries for WebAssembly.

In this chapter we describe our technical contributions. In Section 3.1 we enunciate how the current state-of-the-art lead us to contribute with Software Diversification through LLVM. We follow by describing our two contributions and their main technical insights in Section 3.2 and Section 3.3. Besides, we describe a new transformation strategy as part of our contributions.

3.1 Artificial Software Diversity for WebAssembly

The work of Hilbig et al. [5] at 2021 influences our engineering decisions. According to their work LLVM-based compilers created the 70% of the WebAssembly binaries in the wild. Therefore, we decided to provide Artificial Sotfware Diversity for WebAssembly through LLVM. Other solutions would have been to diversify at the source code level, or at the WebAssembly binary level. However, the former would limit the applicability of our work. The latter, will be addressed in future works (Section 6.2).

LLVM is composed by three main components [?] . First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation passes improve the LLVM IR. Third and final, the backend component is in charge of generating the target machine code. Notice that, the LLVM architecture is highly scalable. The machine code translation of LLVM IR might have any number of custom intermediate passes. In Figure 3.1 we show the generic workflow followed in our contributions. In the context of our work the LLVM architecture is instantiated over all LLVM frontends (1), it adds a Diversifier as a LLVM IR pass (2) and uses a custom Wasm backend (3). The dashed squares in Figure 3.1 wrap the components for which we contribute.

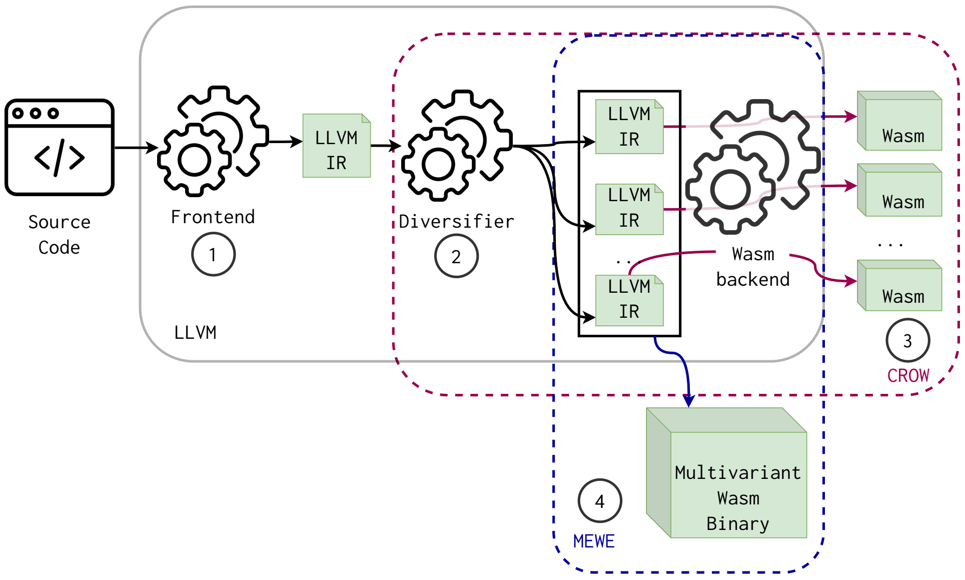


Figure 3.1: Generic workflow to create WebAssembly program variants.

The generic workflow in Figure 3.1 starts by receiving the source code from the program to be compiled to WebAssembly . Then, an LLVM frontend transforms this source code into LLVM IR representation. The resulting LLVM IR is the input for the Diversifier (1). The diversifier generates LLVM IR variants from of the output of the frontend (2). These variants are inputs for our customized Wasm backend. In our case the LLVM backend is always for WebAssembly and, it finalizes the creation of the variants (3).

Our first technical contribution, CROW [8], includes the implementation of the diversifier for LLVM and the customized WebAssembly backend. CROW is able to create several WebAssembly program variants out of a source code program. In Section 3.2 we dissect CROW into more details. In addition, an orthogonal contribution comes from the generation of LLVM IR variants at Step (2). Our second contribution, MEWE [7], merges and creates multivariant binaries to provide MVE for WebAssembly (4). In Section 3.3 we describe MEWE in details.

3.2 CROW: Code Randomization Of WebAssembly

This section describes CROW [8], our first contribution. Following the workflow in Figure 3.1, CROW is a tool tailored to create semantically equivalent WebAssembly variants out of LLVM IR passed through the LLVM frontend to be compiled as WebAssembly code. In Figure 3.2, we describe the architecture of CROW to create program variants. The figure highlights the main two components of the Diversifier, *exploration* and *combining*. The workflow starts by passing the input LLVM IR to perform the *exploration*. During the *exploration* process, at the instruction level for each function in the input LLVM IR, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM bitcode variants. Then, the corresponding backend compiles the LLVM IR variants into WebAssembly binaries.

In the following, we describe our engineering decisions. All our implementation choices are based on one premise: each implementation decision should increase the number of WebAssembly variants that CROW creates.

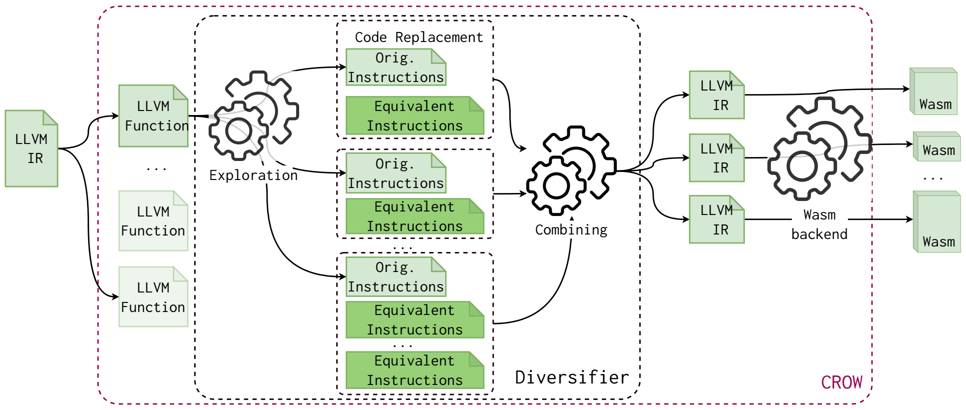


Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM bitcodes to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

Exploration

The key component of CROW if its code replacements generation strategy. The diversifier implemented in CROW is based on the work of Jacob et al. [52]. Their work uses code superoptimization to generate software diversification for x86 with an approach called superdiversification. Jacob and colleagues cornerstone, code superoptimization, focuses on *searching* for a new program which is faster or smaller than the original code, while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [67] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. The main difference between the superoptimization process and a superdiversifier it that the latter keeps all intermediate search results despite their performance.

We use the seminal work of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on hand-written rules. Besides, this technique is fully automatic. Second, there is a battle tested superoptimizer for LLVM, Souper [30]. This latter, makes feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. The searching algorithm of Souper is based on inferring the smallest Data Flow Graph for all integer returning instructions in the input LLVM IR. For a given integer returning instruction, it exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original. The search halts when a semantically equivalent expression is found. Souper synthesizes the replacements in increasing size, thus, the first found equivalent transformation is the best and is the result of the searching. Instead of stopping the process as soon the first equivalent transformation is found, we remove it, keeping more equivalent replacements during the searching.

Notice that the searching space exponentially increases with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions that have no correspondence in the WebAssembly backend. This decision reduces the searching space. For example, creating expression having the `freeze` LLVM instructions will increase the searching space for an instruction without a Wasm suitable opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more variants. For example, Souper avoids to construct redundant expressions such as commutative operations. We disable strategies like this for the sake of more statically different programs.

As we discussed in Section 2.2, the equivalence checking is an important part of any program transformation process. In the case of CROW, we quarantee the equivalence property for program variants through Souper as well. Souper uses Z3 [?] , an SMT solver, to check for programs equivalence. Besides, as a sanity check, it uses KLEE and Alive to double-check that the generated LLVM IR binary out of the code replacement is valid.

Constant inferring

By extending Souper as a superdiversifier, we contribute with a new mutation strategy, *constant inferring* (in addition to the before mentioned strategies in Section 2.2). The main component of Souper infers pieces of code as a single constant assignment particularly for boolean valued variables that are used to control branches. If a program branching is removed due to a constant inferring, the generated program is considerably different to the original program, statically and dynamically.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop which stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) % 1000000
            ↪  != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n", current);
8       return 0 ;
9   }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;



    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the `while`-loop because the loop count is too large. Souper can deal with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code, and is removed in the final compilation. It is clear that the new program in Listing 3.2 is remarkably smaller and faster than the original code. Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also notice that the two programs in the example follow the definition of *functional equivalence* discussed in Section 2.2.

Combining replacements

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformation only to the instruction for which it was found in the first place. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

Removing latter optimizations for LLVM

During the implementation of CROW we have the premise of removing all builtin optimizations in the LLVM backend that could reverse Wasm variants. Therefore, in addition to the extension of Souper, we modify the LLVM compiler and the WebAssembly backend. We disable all optimizations in the WebAssembly backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

3.2.1 CROW instantiation

Let us illustrate how CROW works with the simple example code in Listing 3.3. The `f` function calculates the value of $2*x+x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

Listing 3.3: C function that calculates the quantity $2x + x$

```
int f(int x) {
    return 2 * x + x;
}
```

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes 2 + 1 candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program.

Listing 3.4: LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.

```
define i32 @f(i32) {
         code 2
         code 1
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

ret i32 %3
}

define i32 @main() {
%1 = tail call i32 @f(i32
      10)
ret i32 %1
}
```

```
Replacement candidates for
      code_1

%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2

%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.5: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

Listing 3.6: Wasm program variants generated from program Listing 3.3.

```
func $f (param i32) (result i32)
  local.get 0
   i32.const 2
   i32.mul
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 1
   i32.shl
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   local.get 0
   i32.add
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 3
   i32.mul
```

3.3 MEWE: Multi-variant Execution for WEbAssembly

This section describes MEWE [7], our second contribution. The core idea of MEWE is to synthesize diversified function variants, using CROW, providing execution-path randomization in an MVE. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step $\textcircled{2}$ in Figure 3.1 shows. CROW generates each one of these variants with fine-grained diversification at instruction level, applying the majority of the strategies discussed in Section 2.2 and *constant inferring*. Besides, MEWE inlines function variants when appropriate, also resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE($\textcircled{4}$) from the diagram in Figure 3.1. MEWE takes the LLVM IR variants generated by our diversifier and merges them into a Wasm multivariant. In the figure, we highlight the two components of MEWE. In Step $\textcircled{I}$, we merge the LLVM IR variants created by CROW and we create a LLVM multivariant binary. In Step $\textcircled{II}$, we use a special component, called a "Mixer", which augments the binary with a random generator, which is required for performing the execution-path randomization. Also at this stage, the multivariant binary is fixed with the entrypoint of the original binary. The final output of Step $\textcircled{III}$ is a standalone multivariant WebAssembly binary that can be directly deployed. The source code of MEWE can be found at TODO .

Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function that is in charge of choosing a variant

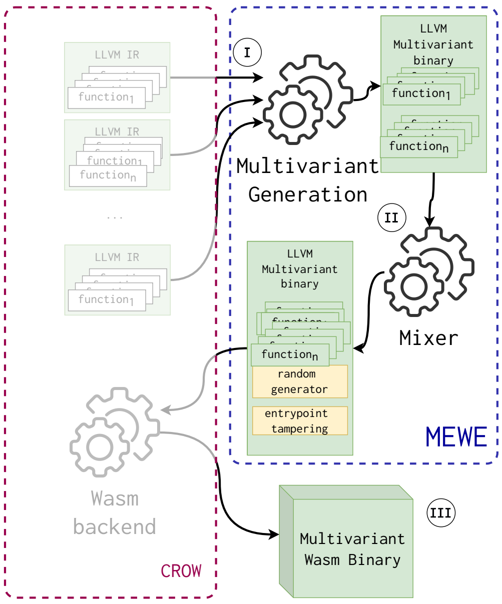


Figure 3.3: Overview of MEWE. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary and then generates a LLVM multivariant binary composed of all the function variants. Also, it includes the dispatcher functions in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and a tampering of the original application entrypoint, in order to produce a WebAssembly multivariant binary ready to be deployed.

at random, every time the original function is invoked during the execution. With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph $\langle N, E\rangle$ where the nodes in N represent all the functions in the binary and an edge $(f_1, f_2) \in E$ represents a possible invocation of f_2 by f_1 [69], where the nodes are typed. The nodes in N have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In Figure 3.4, we show the original static call graph for and original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original program includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for

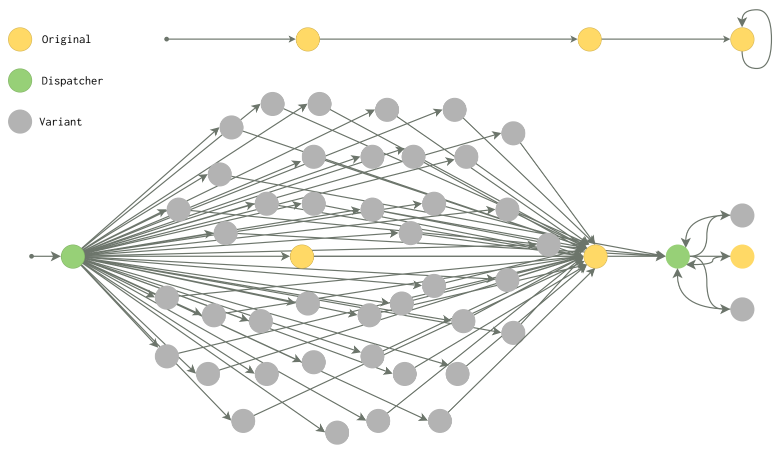


Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value that is then used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [2]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [4]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

Mixer

The Mixer has four specific objectives: tamper the entrypoint of the application, link the LLVM multivariant binary, inject a random generator and merge all these components into a multivariant WebAssembly binary. We use the Rustc compiler[1] to orchestrate the mixing. For the random generator, we rely on

[1]`https://doc.rust-lang.org/rustc/what-is-rustc.html`

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.7: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

WASI's specification [9] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed.

The MEWE mixer creates a new entrypoint for the binary called *entrypoint tampering*. It simply wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

Conclusions

This chapter discusses the technical details for the artifacts implemented for our two contributions. We describe how CROW generates program variants. We introduce a new mutation strategy that is a consequence of retargeting a superoptimizer for LLVM as a superdiversifier. Besides, we dissect MEWE and how it creates an MVE system. In Chapter 4 we discuss the methodology we follow to evaluate the impact of CROW and MEWE.

Chapter 4

Methodology

In this chapter, we present our methodology to answer the research questions enunciated in Section 1.2. We investigate three research questions. In the first question, **TODO** we artificially generate WebAssembly program **TODO** This is the how, the why is more important variants and quantitatively compare the static differences between variants. Our second research question focuses on comparing their behavior during their execution. The final research question evaluates the feasibility of using the program variants in security-sensitive environments. We evaluate our generated program variants in an Edge-Cloud computing platform proposing a novel multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use **TODO** too generic: the empirical method [13] , proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

Corpora

Our experiments assess the impact of artificially created diversity. The first step is to build a suitable corpus of programs' seeds to generate the variants. Then, we answer all our research questions with three corpora of diverse and representative programs for our experiments. **TODO** elaborate on diverse and representative We build **TODO** why three: tell the reader here it feels like a magic number our three corpora in an escalating strategy. The first corpus is diverse and contains simple programs in terms of code size, making them easy to manually analyze.

The second corpus is a project meant for security-sensitive applications. The third corpus is a QR encoding decoding algorithm. The work of Hilbig et al. [5] shows that approximately 65% of all WebAssembly programs come out of C/C++ source code through the LLVM pipeline, and more than 75% if the Rust language is included. Therefore, all modules in the corpora are considered to come along the LLVM pipeline. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** : We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library implemented in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Besides, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In Table 4.1 we listed the corpus name, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus.
TODO Add commit sha1 for libsodium and qrcode.

[1]http://www.rosettacode.org/wiki/Rosetta_Code

Corpus	No. modules	No. functions	LOC range	Location
Rosetta	-	303	7 - 150	`https://github.com/KTH/slumps/tree/master/benchmark_programs/rossetta/valid/no_input`
Libsodium	102	869	8 - 2703	`https://github.com/jedisct1/libsodium`
QrCode	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust`
Total		3021		

Table 4.1: Corpora description. The table is composed by the name of the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

4.1 RQ1. To what extent can we artifically generate program variants for WebAssembly ?

TODO the main issue with "Methoology for RQ1" is that CROW has not been introcuded / cast as a contribution yet

This research question investigates whether we can artificially generate program variants for WebAssembly . We use CROW to generate variants from an original program, written in C/C++ in the case of Rosetta corpus and LLVM bitcode modules in the case of Libsodium and QrCode. In Figure 4.1 we illustrate the workflow to generate WebAssembly program variants. We pass each function of the corpora to CROW as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated WebAssembly variants.

Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two values to quantitatively evaluate RQ1 at the corpus level.

We start by defining what a program's population is. This definition can be applied in general to any collection of variants of the same program. All definitions are based upon bytecodes and not the source code of the programs.

Definition 2. *Program's population $M(P)$:* Given a program P and its generated variants v_i, the program's population is defined as.

$$M(P) = \{v_i \text{ where } v_i \text{ is a variant of P}\}$$

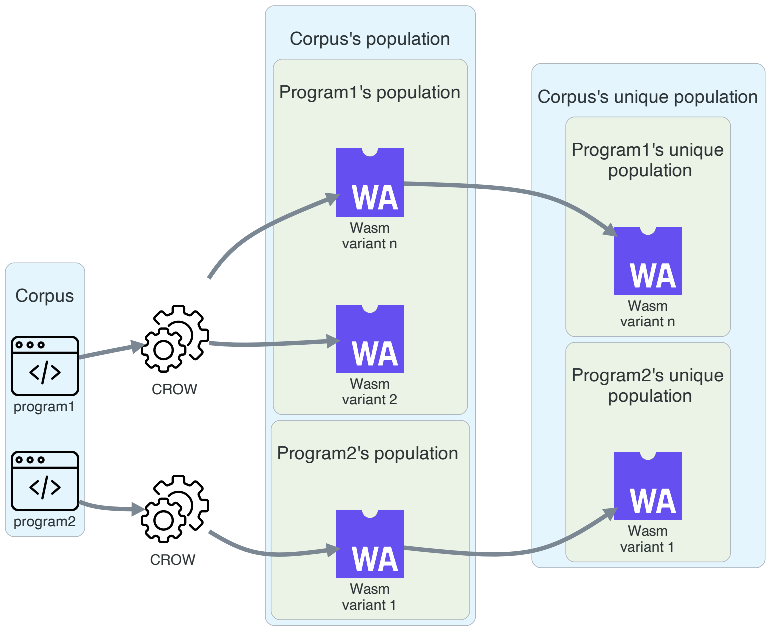


Figure 4.1: The program variants generation for RQ1.

Notice that, the program's population includes the original program P.

Beyond the program's population, we also want to compare how many program variants are unique. The subset of unique programs in the program's population hints how the variants are different between them and not only against the original program. For example, imagine a program P with two program variants V_1 and V_2, the program population is composed by $\{P, V_1$ and $V_2\}$ where V_1 is different from P, and V_2 is different from P. Either, if V_1 is equal or different from V_2, the program's population still be the same.

Definition 3. *Program's unique population* $U(P)$*:* Given a program P and its program's population $M(P)$, the program's unique population is defined as.

$$U(P) = \{v \ \in \ M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $md5sum(v_i) \neq md5sum(v_j)$. $Md5sum(v)$ is the md5 hash calculated over the bytecode stream of the program file v. Notice that, the original program P is included in $U(P)$.

Metric 1. *Program's population size $S(P)$:* Given a program P and its program's population $M(P)$ according to Definition 2, the program's population size is defined as.

$$S(P) = |M(P)|$$

Metric 2. *Program's unique population size $US(P)$:* Given a program P and its program's unique population $U(P)$ according to Definition 3, the program's unique population size is defined as.

$$US(P) = |U(P)|$$

Metric 3. *Corpus population size $CS(C)$:* Given a program's corpus C, the corpus population size is defined as the sum of all program's population sizes over the corpus C.

$$CS(C) = \Sigma S(P) \; \forall \; P \; \in \; C$$

Metric 4. *Corpus unique population size $UCS(C)$:* Given a program's corpus C, the corpus unique population size is defined as the sum of all program's unique population sizes over the corpus C

$$UCS(C) = \Sigma US(P) \; \forall \; P \; \in \; C$$

Protocol

To generate program variants, we synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [24] against the original program. **TODO** link with SOTA . For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all the 60 supported instructions in our synthesizer.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and QrCode is motivated by the properties listed in Table 4.1. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the 303 + 869 + 1849 functions in the corpora to CROW, as Figure 4.1 illustrates, to synthesize program variants. We calculate the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus and conclude by answering RQ1.

4.2 RQ2. To what extent are the generated variants dynamically different?

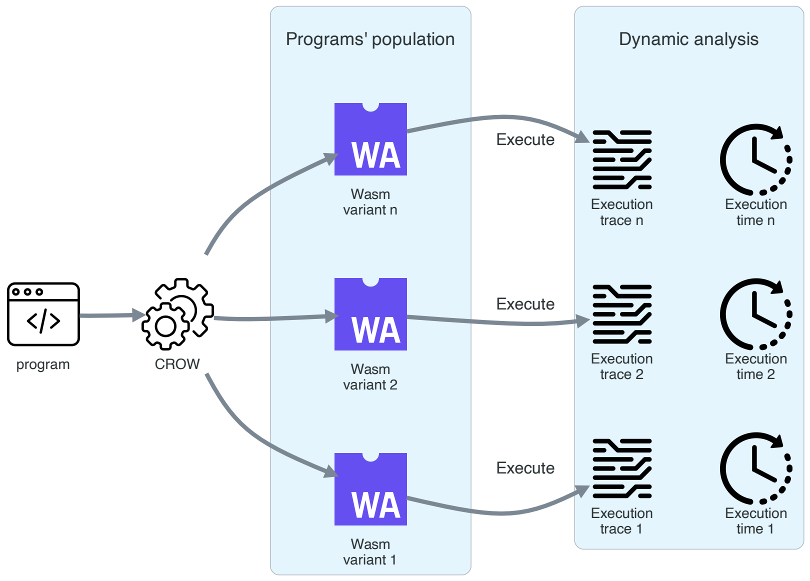


Figure 4.2: Dynamic analysis for RQ2.

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate our experiments into two fields as Figure 4.2 illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program, and we address it in answering RQ1. With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the

original functions from Rosetta corpus described in Section 4 and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the WebAssembly binary to it.
TODO Motivate, Increasing attack surface does not necesarilly has an impact on defence. For example, reordering instructions for code blocks that are never executed does not impact attacker

To dynamically compare programs and their variants, we execute each program on each programs' population to collect and execution times. We define execution trace and execution time in the following section.

Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [70] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. DTW has been used for comparing traces in different domains. For software, De A. Maia et al. [51] proposed to identify similarity between programs from execution traces. In our experiments, we define the traces as the sequence of the stack operations during runtime, *i.e.,* the consecutive list of `push` and `pop` operations performed by the WebAssembly engine during the execution of the program. In the following, we define the $TraceDiff$ metric.
TODO before, define this and give an illutrative listing plus, says how you collect those traces, that's part of the protocol: between the stack operation traces

Metric 5. *TraceDiff:* Given two programs P and P' from the same program's population, TraceDiff(P,P'), computes the DTW distance collected during their execution.
A TraceDiff of 0 means that both traces are identical. The higher the value, the more different the traces.

Moreover, we use the execution time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution time distributions. We define the execution time as follows:

Metric 6. *Execution time:* Given a WebAssembly program P, the execution time is the time spent to execute the binary.

Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (Metric 5). We use SWAM[2] to execute each program and variant to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations.

Furthermore, we collect the execution time, Metric 6, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Withney U test [72] in a pairwise strategy.

4.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

TODO The last method is too short

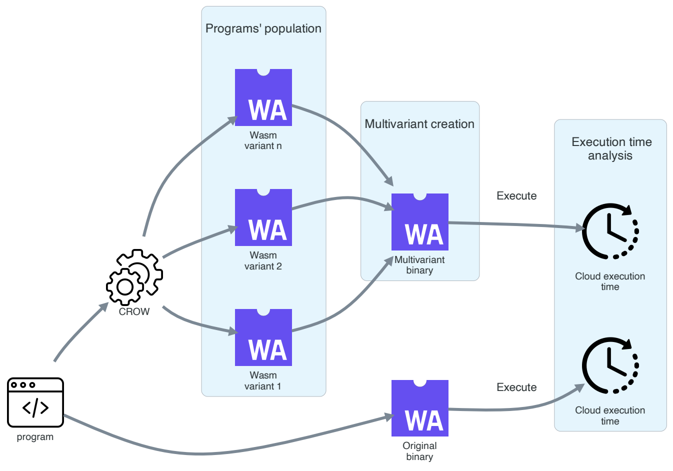


Figure 4.3: Multivariant binary creation and workflow for RQ3 answering.

TODO Motivate the usage of Edge based platforms...

In the last research question, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries (according to Definition 4), and we deploy and execute them at the Edge. The process of *mixing* multiple variants into one multivariant binary is an essential contribution of the thesis that is presented in details in [6]. RQ3 focuses on analyzing the impact of

[2]`https://github.com/satabin/swam`

this contribution on execution times and timing side-channels. To answer RQ3, we use the variants generated for the programs of Libsodium and QrCode corpora, we take $2 + 5$ programs interconnecting the LLVM bitcode modules (mentioned in Table 4.1). We illustrate the protocol to answer RQ3 in Figure 4.3 starting from the creation of the programs' population.

Metrics

To answer RQ3, we build multivariant WebAssembly binaries meant to provide path execution diversification. In the following we define what a multivariant binary is.

Definition 4. *Multivariant binary:* Given a program P with functions $\{F_1, F_2, ..., F_n\}$ for which we generate function's populations (according to Definition 2) $\{M(F_1), M(F_2), ..., M(F_n)\}$. A multivariant binary B is the composition of the functions populations following the function call graph of P where each call $F_i \rightarrow F_j$ is replaced by a call $f \rightarrow g$, where $f \in M(F_i)$ and $g \in M(F_j)$ are function variants randomly selected at runtime.

We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in Metric 6 for the execution time of multivariant binaries.

Protocol

We answer RQ3 by analyzing a real-world scenario. Since Wasm binaries are currently adopted for Edge computing, we run our experiments for RQ3 on the Edge. Edge applications are designed to be deployed as isolated HTTP services, having one single responsibility that is executed at every HTTP request. This development model is known as serverless computing, or function-as-a-service [12, 2]. We deploy and execute the multivariant binaries as end-to-end HTTP services on the Edge and we collect their execution times. To remove the natural jitter in the network, the execution times are measured at the backend space, *i.e.,* we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header.

We do this process twice, for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world. In Figure 4.4 we illustrate the word wide location of the edges nodes.

We collect 100k execution times for each binary, both the original and multivariant binaries. The number of execution time samples is motivated by the seminal work of Morgan et al. [34]. We perform a Mann-Withney U test [72] to compare both execution time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

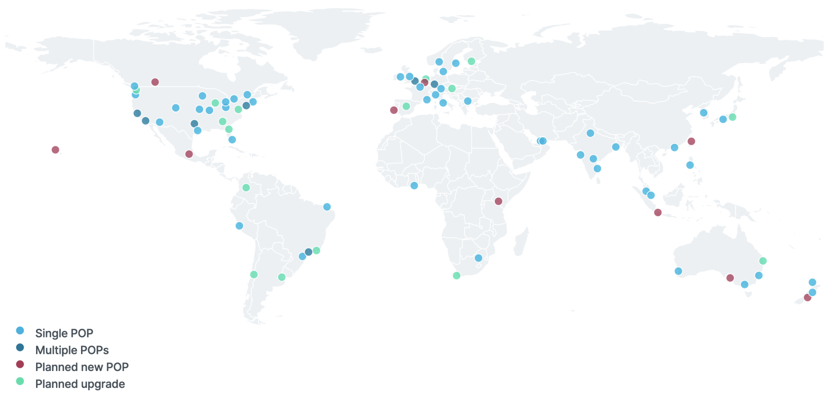


Figure 4.4: Screenshot taken from the Fastly Inc. platform used in our experiments for RQ3. Blue and darker blue dots represent the edge nodes used in our experiments.

Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Besides, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.

Chapter 5

Results

In this chapter, we sum up the results of the research of this thesis. We illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 4.

5.1 RQ1. To what extent can we artifically generate program variants for WebAssembly ?

As we describe in Section 4.1, our first research question aims to answer how to artifically generate WebAssembly program variants. This section is organized as follows. First we present the general results calculating the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

Program's populations

We summarize the results in Table 5.1. The table illustrates the corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant), the cumulative number of variants (*Corpus population size*) and the cumulative number of unique variants (*Corpus unique population size*).

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a diversification timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, representing a 11.78% of the total. As the four and fifth columns show, the number of artifically created variants and the number of unique variants are larger than the original number of functions by one order of magnitude. In the case of Rosetta , the corpus population size is close to one million of programs.

TODO Elaborate on uniqueness...

TODO LOW: add histogram on variant sizes

Corpus	#Functions	# Diversified	# Variants	# Unique Variants
Rosetta	303	239	809900	2678
Libsodium	869	85	4272	3805
QrCode	1849	32	6369	3314
	3021	356	820541	9797

Table 5.1: General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.

Challenges for automatic diversification

We have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of Table 5.1). Our approach successfully diversified 239/303, 85/869 and 32/1849 of the original functions for Rosetta , Libsodium and QrCode respectively. The main reason of this phenomenon is the set timeout for CROW.

We have noticed a remarkable difference between the number of diversified functions for each corpus, 809900 programs for Rosetta 4272 for Libsodium and 6369 for QrCode. The corpus population size for Rosetta is two orders of magnitude larger compared to the other two corpora. The reason behind the large number of variants for Rosetta is that, after certain time, our approach starts to combine the code replacements to generate new variants. However, looking at the fifth column, the number of unique variants have the same order of magnitude for all corpora. The variants generated out of the combination of several code replacements are not necessarily unique. Some code replacements can dominate over others, generating the same WebAssembly programs.

A low timeout offers more unique variants compared to the population size despite the low number of diversified functions, like the Libsodium and QrCode cases. This happens because, CROW first generates variants out of single code replacements and then starts to combine them. Thus, more unique variants are generated in the very first moments of the diversification process with CROW.

Apart from the timeout and the combination of variants phenomena, we manually analyze programs, searching for properties attempting to the generation of program variants using CROW. We identify another challenge for diversification. We have observed that our approach searches for a constant replacement for more than 45% of the instructions of each function while constant values cannot be inferred. For instance, **TODO** should be first briefly defined for the reader: constant values cannot be inferred. can you elaborate on the importance of constant values for diversification. for memory load operations because our tool is oblivious to a memory model.

Properties for large diversification

We manually analyzed the programs to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 5.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 5.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 5.3), using the intrinsics of the underlying computation model.

Listing 5.1: Diversification through arithmetic expression replacement.

```
local.get 0       local.get 0
i32.const 2       i32.const 1
i32.mul           i32.shl
```

Listing 5.2: Diversification through inversion of comparison operations.

```
local.get 0       i32.const 11
i32.const 10      local.get 0
i32.gt_s          i32.le_s
```

Listing 5.3: Diversification through overflow of integer operands.

```
i32.const 2       i32.const 2
i32.mul           i32.mul
                  i32.const
                    -2147483647
                  i32.mul
```

TODO Add an example and refer to Cohen when machine code is generated, show that jumps are changed

Answer to RQ1.

We can provide diversification for 11.78% of the programs in our corpora. Constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants.

5.2 RQ2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated to answer RQ1 for Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (Metric 5) and the execution time distributions (Metric 6) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with TraceDiff of Metric 5. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution time distributions Metric 6, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

Stack operation traces.

In Figure 5.1 we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (Metric 5) for a program of Rosetta corpus for which we generate variants. Each dot represents a comparison between two programs' traces according to Metric 5. The programs are sorted by their number of variants in descending order. For the sake of illustration, we filter out those programs for which we generate only 2 unique variants.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of TraceDiff is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.

In some cases, we have observed variants that are statically different for which TraceDiff value is zero, *i.e.,* they result in the same stack operation trace. We

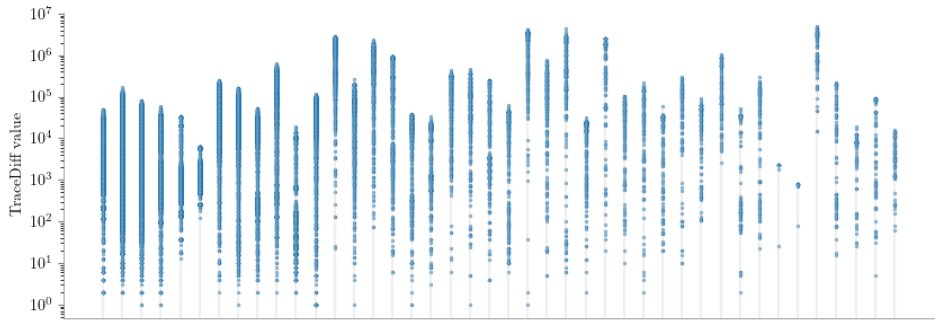


Figure 5.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 5.

identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

```
(1) i32.lt_u        i32.lt_s        (3) i32.ne           i32.lt_u
(2) i32.le_s        i32.lt_u        (4) local.get 6      local.get 4
```

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

Execution times.

TODO recall again that execution time diversification is important, should the reader have forgotten about this.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of WebAssembly programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution environment. This mentioned reasoning enforces our comparison of the execution times for the generated variants. For each program's population, we compare the execution time distributions, Metric 6, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a different execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for one programs' population of Rosetta . The plots in Figure 5.2 show the execution time distributions for the `Hilbert curve` program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in Section 5.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The plot is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in Section 5.3.

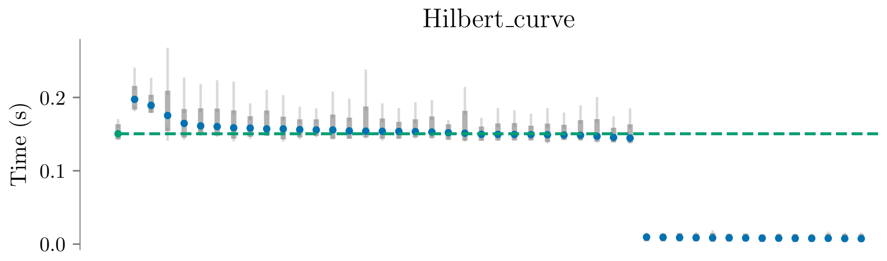


Figure 5.2: Execution time distributions for `Hilber_curve` program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.

Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. For example, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a resilient execution.

5.3 RQ3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for Libsodium and QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

Execution times and timing side-channels.

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 5.3, each subplot represents the quantile-quantile plot [71] of the two distributions, original and multivariant binary. This kind of plots is used to compare the shapes of distributions, providing a graphical comparison of location, scale, and skewness for two distributions. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.,* for two equal distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others.

Answer to RQ3.

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. Consequently, attacks relying on measuring precise execution times [**?**] of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

Conclusions

Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of programs by 4.15 times. We highlighted the importance of complementing static and dynamic studies for programs diversification. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a given query. We empirically demonstrate the feasibility and the application of automatically generating WebAssembly program variants.

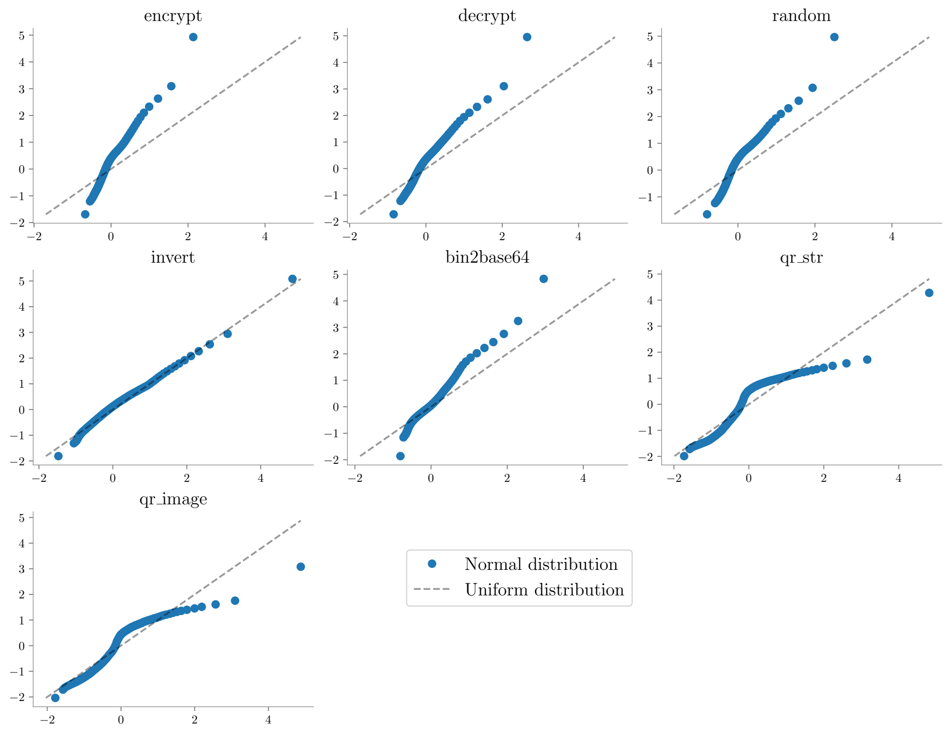


Figure 5.3: Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

Chapter 6

Conclusion and Future Work

6.1 Summary of the results

6.2 Future work

6.2.1 wasm-mutate future work

Bibliography

[1] Romano,A., Lehmann,D., Pradel,M., and Wang,W. (2022). Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1101–1116, Los Alamitos, CA, USA. IEEE Computer Society.

[2] Narayan,S., Disselkoen,C., Moghimi,D., Cauligi,S., Johnson,E., Gang,Z., Vahldiek-Oberwagner,A., Sahita,R., Shacham,H., Tullsen,D., et al. (2021). Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*.

[3] Lee,S., Kang,H., Jang,J., and Kang,B. B. (2021). Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*.

[4] Johnson,E., Thien,D., Alhessi,Y., Narayan,S., Brown,F., Lerner,S., McMullen,T., Savage,S., and Stefan,D. (2021). Sfi safety for native-compiled wasm. *NDSS. Internet Society*.

[5] Hilbig,A., Lehmann,D., and Pradel,M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*.

[6] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021b). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[7] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021a). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[8] Cabrera Arteaga,J., Floros,O., Vera Perez,O., Baudry,B., and Monperrus,M. (2021). Crow: code diversification for webassembly. In *MADWeb, NDSS 2021*.

[9] (2021). Webassembly system interface.

[10] Xu,Y., Solihin,Y., and Shen,X. (2020). Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*, pages 987–1000.

[11] Tsoupidi,R. M., Lozano,R. C., and Baudry,B. (2020). Constraint-based software diversification for efficient mitigation of code-reuse attacks. *ArXiv*, abs/2007.08955.

[12] Shillaker,S. and Pietzuch,P. (2020). Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*, pages 419–433.

[13] Runeson,P., Engström,E., and Storey,M.-A. (2020). *The Design Science Paradigm as a Frame for Empirical Software Engineering*, pages 127–147. Springer International Publishing, Cham.

[14] Lehmann,D., Kinder,J., and Pradel,M. (2020). Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[15] Harrand,N., Soto-Valero,C., Monperrus,M., and Baudry,B. (2020). Java decompiler diversity and its application to meta-decompilation. *Journal of Systems and Software*, 168:110645.

[16] Gadepalli,P. K., McBride,S., Peach,G., Cherkasova,L., and Parmer,G. (2020). Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, page 265–279.

[17] Chen,D. and W3C group (2020). WebAssembly documentation: Security. Accessed: 18 June 2020.

[18] Bryant,D. (2020). Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*, pages 9–12.

[19] Österlund,S., Koning,K., Olivier,P., Barbalace,A., Bos,H., and Giuffrida,C. (2019). kmvx: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[20] Churchill,B., Padon,O., Sharma,R., and Aiken,A. (2019). Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA. Association for Computing Machinery.

[21] Aga,M. T. and Austin,T. (2019). Smokestack: thwarting dop attacks with runtime stack layout randomization. In *Proc. of CGO*, pages 26–36.

[22] Silvanovich,N. (2018). The problems and promise of webassembly. Technical report.

[23] Lu,K., Xu,M., Song,C., Kim,T., and Lee,W. (2018). Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*.

[24] Li,J., Zhao,B., and Zhang,C. (2018). Fuzzing: a survey. *Cybersecurity*, 1(1):1–13.

[25] Gupta,S., Saxena,A., Mahajan,A., and Bansal,S. (2018). Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In Beyersdorff,O. and Wintersteiger,C. M., editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 365–382, Cham. Springer International Publishing.

[26] Genkin,D., Pachmanov,L., Tromer,E., and Yarom,Y. (2018). Drive-by key-extraction cache attacks from portable code. *IACR Cryptol. ePrint Arch.*, 2018:119.

[27] Belleville,N., Couroussé,D., Heydemann,K., and Charles,H.-P. (2018). Automated software protection for the masses against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 15(4).

[28] WebAssembly Community Group (2017b). WebAssembly Specification.

[29] WebAssembly Community Group (2017a). WebAssembly Roadmap.

[30] Sasnauskas,R., Chen,Y., Collingbourne,P., Ketema,J., Lup,G., Taneja,J., and Regehr,J. (2017). Souper: A Synthesizing Superoptimizer. *arXiv preprint 1711.04422.*

[31] Haas,A., Rossberg,A., Schuff,D. L., Schuff,D. L., Titzer,B. L., Holman,M., Gohman,D., Wagner,L., Zakai,A., and Bastien,J. F. (2017). Bringing the web up to speed with webassembly. *PLDI.*

[32] Van Es,N., Nicolay,J., Stievenart,Q., D'Hondt,T., and De Roover,C. (2016). A performant scheme interpreter in asm.js. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1944–1951, New York, NY, USA. Association for Computing Machinery.

[33] Couroussé,D., Barry,T., Robisson,B., Jaillon,P., Potin,O., and Lanet,J.-L. (2016). Runtime code polymorphism as a protection against side channel attacks. In *IFIP International Conference on Information Security Theory and Practice*, pages 136–152. Springer.

[34] Morgan,T. D. and Morgan,J. W. (2015). Web timing attacks made practical. *Black Hat.*

[35] Kim,D., Kwon,Y., Sumner,W. N., Zhang,X., and Xu,D. (2015). Dual execution for on the fly fine grained execution comparison. *SIGPLAN Not.*

[36] Davi,L., Liebchen,C., Sadeghi,A.-R., Snow,K. Z., and Monrose,F. (2015). Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS.*

[37] Crane,S., Homescu,A., Brunthaler,S., Larsen,P., and Franz,M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11.

[38] Baudry,B. and Monperrus,M. (2015). The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1).

[39] Alon Zakai (2015). asm.js Speedups Everywhere.

[40] Agosta,G., Barenghi,A., Pelosi,G., and Scandale,M. (2015). The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1320–1333.

[41] Le,V., Afshari,M., and Su,Z. (2014). Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226.

[42] Homescu,A., Neisius,S., Larsen,P., Brunthaler,S., and Franz,M. (2013). Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE.

[43] Maurer,M. and Brumley,D. (2012). Tachyon: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 617–630.

[44] Jackson,T. (2012). *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine.

[45] Cleemput,J. V., Coppens,B., and De Sutter,B. (2012). Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4).

[46] Sidiroglou-Douskos,S., Misailovic,S., Hoffmann,H., and Rinard,M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 124–134, New York, NY, USA. Association for Computing Machinery.

[47] Jackson,T., Salamat,B., Homescu,A., Manivannan,K., Wagner,G., Gal,A., Brunthaler,S., Wimmer,C., and Franz,M. (2011). Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer.

[48] Amarilli,A., Müller,S., Naccache,D., Page,D., Rauzy,P., and Tunstall,M. (2011). Can code polymorphism limit information leakage? In *IFIP International Workshop on Information Security Theory and Practices*, pages 1–21. Springer.

[49] Chen,T. Y., Kuo,F.-C., Merkel,R. G., and Tse,T. H. (2010). Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66.

[50] Salamat,B., Jackson,T., Gal,A., and Franz,M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46.

[51] Maia,M. D. A., Sobreira,V., Paixão,K. R., Amo,R. A. D., and Silva,I. R. (2008). Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, pages 6–10.

[52] Jacob,M., Jakubowski,M. H., Naldurg,P., Saw,C. W. N., and Venkatesan,R. (2008). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer.

[53] de Moura,L. and Bjørner,N. (2008). Z3: An efficient smt solver. In Ramakrishnan,C. R. and Rehof,J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[54] Salamat,B., Gal,A., Jackson,T., Manivannan,K., Wagner,G., and Franz,M. (2007). Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor. Technical report, Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[55] Bruschi,D., Cavallaro,L., and Lanzi,A. (2007). Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference.*

[56] Younan,Y., Pozza,D., Piessens,F., and Joosen,W. (2006). Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438.

[57] Cox,B., Evans,D., Filipi,A., Rowanhill,J., Hu,W., Davidson,J., Knight,J., Nguyen-Tuong,A., and Hiser,J. (2006). N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium*, USENIX-SS'06.

[58] Pohl,K., Böckle,G., and Van Der Linden,F. (2005). *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer.

[59] Bhatkar,S., Sekar,R., and DuVarney,D. C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*, pages 271–286.

[60] El-Khalil,R. and Keromytis,A. D. (2004). Hydan: Hiding information in program binaries. In Lopez,J., Qing,S., and Okamoto,E., editors, *Information and Communications Security*, pages 187–199, Berlin, Heidelberg. Springer Berlin Heidelberg.

[61] Kc,G. S., Keromytis,A. D., and Prevelakis,V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*, pages 272–280.

[62] Bhatkar,S., DuVarney,D. C., and Sekar,R. (2003). Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium.*

[63] Barrantes,E. G., Ackley,D. H., Forrest,S., Palmer,T. S., Stefanovic,D., and Zovi,D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*, pages 281–289.

[64] Chew,M. and Song,D. (2002). Mitigating buffer overflows by operating system randomization. Technical Report CS-02-197, Carnegie Mellon University.

[65] Cohen,F. B. (1993). Operating system protection through program evolution. *Computers & Security*, 12(6):565–584.

[66] Pettis,K. and Hansen,R. C. (1990). Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 16–27, New York, NY, USA. Association for Computing Machinery.

[67] Henry,M. (1987). Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126.

[68] Avizienis and Kelly (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80.

[69] Ryder,B. G. (1979). Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226.

[70] Needleman,S. B. and Wunsch,C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. 48(3):443–453.

[71] Gnanadesikan,R. and Wilk,M. B. (1968). Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17.

[72] Mann,H. B. and Whitney,D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60.

Index